Quick (unformal) Memo
for (a subset of) Objective Caml

Revision: 1.18 ß Date: 2010/04/21 14:48:35
Philippe.Wang@lip6.fr
http://www-apr.lip6.fr/~pwang/memo_ocaml.pdf

This memo is about a subset of the Objective Caml language. Its purpose is to help you remember what you knew. If you think you are learning from it, please refer to the official documentation to verify the information, at `http://caml.inria.fr`.

Lexical conventions

• **Variables** : `[a-z_][A-Za-z_0-9]*`
• **Types** : `[a-z_][A-Za-z_0-9]*`
• **Type variables** : `'[a-z][A-Za-z_0-9]*`
• **Variant constructors** : `[A-Z][A-Za-z_0-9]*`
• **Record fields** : `[a-z_][A-Za-z_0-9]*`
• **Module names** : `[A-Z][A-Za-z_0-9]*`
• **Module types** (signatures) : `[A-Za-z][A-Za-z_0-9]*`
• **Integer prefixes** : `0[Xx], 0[oO], 0[bB]`
• **Integers** : `[0-9][0-9_]*` (add `a-fA-F` for base-16)
• **Floats** : `[0-9]*.[0-9]*([Ee][0-9]+)?`
• **Booleans** : `(true|false)`
• **Characters** : `'c'` where c is whether an ASCII character, or `'\xAA'` where `00` $\leq$ `AA` $\leq$ `FF`, or `'\222'` where `000` $\leq$ `222` $\leq$ `255`, or `'\t'`, `'\n'`, `'\r'`, `'\b'`.
• **String** : `"str"` (like char for non ASCII bytes + \").
• Any name can be prefixed by the module which contains them with a dot between the prefix and the name, e.g. `String.t` (type `t` of module `String`), `List.map` (value `map` of module `List`), `Map.Make` (module `Make` inside module `Map`), etc.

Keywords
and as assert asr begin class constraint do done downto else end exception external false for fun function functor if in include inherit initializer land lazy let lor lsl lsr lxor match method mod module mutable new object of open or private rec sig struct then to true try type val virtual when while with

```
!=    #     &     &&    '     (     )     *     +     ,     -
-.    ->    .     ..    :     ::    :=    :>    ;     ;;    <
<-    =     >     >]    >}    ?     ??    [     [<    [>    [|
]     _     `     {     {<    |     |]    }     ~
```

• Some keywords can be redefined (e.g. `lsr`, `&&`, `!=`) but shouldn't be: their syntactic behaviour are hard-implemented (and obey no standard rule). • Operators which are not keywords :

Type definitions

• Alias: `type t1 = t2`
• Variant (or Sum or Algebraic data type):
`type s = A | B | C of s`
• Variant definitions are lists of variant constructors only.
• Record: `type r = {label1: t1 ; l2: s}`
• Records contain labels definition only.
• Field names can be prefixed by keyword `mutable` to make their values mutable.
• Types can be parameterized: `type 'a x = X of 'a` or `type ('x,'y) z = {x: 'x ; mutable y: 'y}`. Type variables (such as `'a`, `'b`, `'c`, etc) can be used in a definition only if they appear in the parameters.
• Type definitions are always recursive if need be (e.g. `type 'a t = Nil | Cons 'a * 'a t`).
• Mutual recursion is obtained with keyword `and` (e.g. `type t1 = A of t2 and t2 = B of t1 | C`).
• Some cyclic definitions are rejected.

Patterns

• A pattern can be whether a variable name (`[a-z_][a-zA-Z0-9_]`) which makes a **new** link without any constraint, or a number (`int`, `float`, `Int32.t`, `Int64.t`), or a `char` value, or a `string` value, or a list (`[]` or `[x;y;z]` or `x::y::z::[]`), or a variant constructor name (`[A-Z][a-zA-Z0-9_]`), or a complex pattern such as `Some([1;2;3])`.
• Variant constructors and complex patterns can be linked for further use, with keyword `as`: `(Some([1;x;y;(12345 as z)])) as t`.
• Patterns must be type coherent : `[1;1.2;t]` is not a valid pattern because there cannot be a list containing both an `int` value and a `float` value at the same level.
• `{x=42;y=y} as v` is a valid pattern if there exists a record type `t` with `x` as an int field and `y` as a field (no constraint on its type here); `v` has type `t` an represents the whole value; `y` is `v.y`.

Expressions

• `if expr1 then expr2 else expr3` where `expr1` has type `bool` and `expr2` and `expr3` share the same type; if `expr2` or `expr3` are sequences, they must be parenthesized; has type of `expr2`.
• `while expr1 do expr2 done` where `expr1` has type `bool` and `expr2` should have type `unit`; has type `unit`.
• `for i = expr1 to expr2 do expr3 done` where `i` is a valid variable name, `expr1` and `expr2` have type `int`, and `expr3` should have type `unit`; `expr1` and `expr2` always evaluated whereas `expr3` evaluated only if `expr1 <= expr2`; has type `unit`.
• `for i = expr1 to expr2 downto expr3 done` like `for` loop but `expr3` evaluated only if `expr1 >= expr2`.
• `expr1; expr2` where `expr1` should have type `unit`; has type of `expr2`.
• `(expr1; expr2)` where `expr1` should have type `unit`; has type of `expr2`.
• `match expr1 with | P1 -> expr2 | P2 -> expr3` where `P1` and `P2` are patterns of same type `t1`, `expr1` has type `t1`; `expr2` and `expr3` have same type `t2`; (`P2` can be a variable name instead of a pattern); has type `t2`.
• `fun a -> expr` where `a` is a variable name; has type `t1 -> t2` where `a:t1` and `expr:t2`.
• `fun a b -> expr` where `a` is a variable name; has type `t1 -> t2 -> t3` where `a:t1`, `b:t2` and `expr:t3`.
• `function a -> expr` where `a` is a variable name or a pattern; has type `t1 -> t2` where `a:t1` and `expr:t2`.
• `function | P1 -> expr1 | P2 -> expr2` where `P1` and `P2` are patterns of same type `t1`; `expr1` and `expr2` have same type `t2`; (`P2` can be a variable name instead of a pattern);

has type t1 -> t2.
• a b applies a to b where a has type (t1 -> t2) and b has type t1 ; has type t2 (a and b can be any expression with valid type).
• a b c applies a to b and c where a has type (t1 -> t2 -> t3) and b has type t1, c has type t2 ; has type t3 (a, b and c can be any expression with valid type).
• let v = expr1 in expr2 evaluates expr2 where v links to expr1's evaluation result ; expr1 is evaluated before expr2; has type of expr2.
• try expr1 with EP -> expr2 where expr1 and expr2 have same type t, and EP is a pattern of type exn; has type t. expr2 is evaluated only if an exception is raised by expr1's evaluation and matches pattern EP.
• {contents=x} has type t ref if x has type t.
• x.contents has type t if x has type t ref .
• (expr) ≡ begin expr end.
• (expr1; expr2) ≡ begin expr1; expr2 end.
• v.label is valid if v is a record of type t and label is a field declared in record of type t.
• v.label <- 2 is valid if v.label is valid and label is declared as mutable int.
• (match Some 42 with Some x -> x) returns 42.
• s.[x] ≡ String.get s x.
• s.[x] <- c ≡ String.set s x c.
• "ccc" ≡ String.make 3 'c'.
• a.(x) ≡ Array.get a x.
• a.(x) <- e ≡ Array.set a x e.
• [|x;x|] ≡ Array.make 2 x.

Definitions

• Value : let v = expr where v is a variable name and expr is an expression.
• Function : let f = expr where expr is a function.
• Recursive function or record or a variant: let rec f = expr1 where expr1 is whether a function or a record or a variant; f knows itself.
• Recursive functions or records or a variant: let rec f = expr1 and g = expr2 where expr1 and expr2 are whether functions or records or variants.

Exception definitions

• Exceptions are variant constructors of (the particular extensible) type exn.
• exception E declares an exception E with no argument.
• exception F of float declares an exception F with an argument which has to be of type float.
• All exceptions share the same type (exn) and can be raised with function raise (val Pervasives.raise : exn -> 'a). Function raise does not return, hence the -> 'a.

Top-level

If you write expressions at top-level, make sure you end each top-level definition (of value or type or module or exception) and expression with token ;;. Else there would be syntactic ambiguities, thus errors.

Modules and Functors

module Name = **struct** type t let f x = x **end**
• Contents of file hello.ml are seen outside of hello.ml as contents of module Hello.
• Contents of module M in file hello.ml are seen outside of hello.ml as contents of module Hello.M.
• Access of element E (or e) of module M is M.E (or M.e)
• **module type** T =
sig type defs = .. val f : valtypes .. **end**

Type constraints

• To make sure expression expr's type is compatible with type t, write (expr:t) (**with** the parentheses).
• Write let f : t1 -> t2 = fun x -> expr to make sure f's type is compatible with type t1 -> t2.
• Write module M : T = struct ... end to make sure M's signature is compatible with module type T and to hide elements which are undeclared in T. T can be an inline module type such as sig type t = int val f : t -> t end.
• A functor is parameterized module (there can be several parameters). The parameters types have to be set explicitly.
• module M(P:T)= .. ≡ module M = functor(P:T)-> ..
• module M(P:T)(Q:U)= .. ≡
module M = functor(P:T)->functor(Q:U)-> ..
• Minimal 2-parameter functor:
module M(P:sig end)(Q:sig end) = struct end.
This module M accepts two any unparameterized modules (empty or not) and returns an empty module.

Values

Ranges

type	*plateform*	*range*
int	32bit	$-2^{30}..-2^{30}-1$
int	64bit	$-2^{62}..-2^{62}-1$
Int32.t	any	$-2^{31}..-2^{31}-1$
Int64.t	any	$-2^{63}..-2^{63}-1$
float	any	64bit-precision, IEEE754, 53bit mantissa
char	any	8bit integers, 0..255
string	32bit	vectors of 0 to $2^{24}-5$ char values
string	64bit	vectors of 0 to $2^{57}-9$ char values

Operators

On int values: +, -, *, /, mod, land, lor, lxor, lsl, asr.
On float values: +., -., *., /., **.
On lists: @ (concatenation).
On strings: ^ (concatenation).
Reference: type 'a ref = {mutable contents:'a}
Contents of a reference: let (!) v = v.contents.
Reference assignment: let (:=) a b = a.contents <- b.
Polymorphic test operators: =, <>, <, >, <=, >=.

Value Conversions

To a string value:
string_of_int, string_of_float, string_of_bool,
let string_of_char c = String.make 1 c.
To an int value:
int_of_float, int_of_string, int_of_char.

To a float value:
float_of_int, float_of_string, float_of_char.

Value Printing

```
print_string :  string -> unit
print_int :  int -> unit
print_float :  float -> unit
let print_bool b = print_string(string_of_bool b)
print_newline :  unit -> unit
print_endline :  string -> unit
Printf.printf "%s%d%g" :  string-> int-> float-> unit
Printf.sprintf "%s%d%g" :  string-> int-> float-> string
```

Tips

```
Array.copy:  'a array -> 'a array
let matrix_copy m = Array.map Array.copy m
List.fold_left:('a->'b->'a)->'a->'b list->'a
let sum:int list -> int = List.fold_left (+) 0
```

References

• Objective Caml official web site:
http://caml.inria.fr/ (or http://www.ocaml.org/) • Developing applications with Objective Caml:
http://caml.inria.fr/pub/docs/oreilly-book/ • Développement d'applications avec Objective Caml:
http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/

Feedback

Don't hesitate to send me feedback (in English or in French)
Philippe.Wang@lip6.fr.

